Event-driven Replies to Data Requests

In sync with clock events

Wed, Mar 23, 1994

With the addition of clock event detection hardware in the digital IndustryPack board used in IRMs, we can provide event-driven replies to a data request. For the Classic Protocol, one must be able to specify what clock event should be used to indicate on which 15 Hz cycles the data should be sampled. Using this facility for Linac, for example, one could then reply to a data request only on beam cycles.

The Classic Protocol format for data requests includes the following:

period	#ltypes

The period byte is expressed in cycles (15 Hz), allowing for any period from 1–255 cycles, or 0 for a one-shot request. For event-driven replies, an 8-bit clock event# is needed, so it is natural to specify this in place of the period byte. But then we need to mark the fact that the “period” byte is really an event#. The #listypes byte is usually limited to 4 bits or so, as a Classic request is for a matrix of data to be returned, with all idents processed for each listype#. If 4 bits is enough space for the #listypes field, we can use the upper 4 bits of that byte to contain flags, one of which can mean that the “period” byte is really a clock event#.

When the specified event occurs, an update of the request is generated, so that the data from the data pool is returned on that 15 Hz cycle in which the event has been detected. How can the logic recognize which events have occurred?

The event detecting hardware is programmed to generate an interrupt whenever any event is detected. The interrupt routine reads the event from a FIFO, allowing for many events to occur almost simultaneously without being lost, time-stamps it, and writes the time-stamp into the clock event times table. The information about clock events is present in this table, but it is not so easy to process it in order to quickly check whether it is time to reply to a data request.

One possibility is to maintain a clock event queue, in which is recorded the event# for each event that occurs. The interrupt routine, besides updating the clock event times table, would also write into this queue. To make it easy for a host to read out the contents of this queue, it can be designed as a data stream. But DSWrite, normally used to write records into a data stream, has too much overhead processing for interrupt code, so we can access this queue directly more efficiently.

Now the process of deciding what events have occurred since the last request update is easy. For each request, there must be kept an OUT offset into the event queue that indicates the next record to be checked. Scan all event records written into the queue since the last update, looking for a match on the specified clock event#. If there is a match, then it is time to update and return a reply to the request.

Another approach is to maintain a bit map of events that have occurred since the last cycle. For such requests, logic must be done each cycle to determine whether it is the time to reply. As a result, replies may be updated up to 15 Hz, in the case that the event occurs at 15 Hz. To maintain such a bit map, one must be careful, as events are processed by an interrupt, and the bit map may change between execution of any two instructions. The task-level solution for this is to copy the bit map into another area, then exclusive-OR the copied bit map into the dynamic one. In this way, any bits that were set via interrupts occurring since the bit map was copied are not lost. They will be detected on the next cycle.

A requester of event data could read out the second area at 15 Hz, thus insuring that all events would be noticed. A request slower than 15 Hz would not be able to detect all events, of course, by simply looking at this copied data. If the requester didn’t care about 15 Hz events, but only about Main Ring reset events, say, a new listype ould be designed that gave the bit map at a slower rate, but processing would have to be done at 15 Hz to inclusive-OR the 15 Hz samples of events during those cycles between updates. This may be difficult.

Implementation

Support for event-driven replies has been implemented for Classic Protocol data requests, using the bit map approach internally to detect whether a given event has occurred since the last cycle. In the case that no events of the given type occur, no reply will be generated. This makes it difficult to be sure that the data request was received. The same is true for a server node; it cannot report that a reply is tardy, unless it also knows about the event, too. At the moment, this situation is not detected, but the server node does need to detect the events in order to send replies to the requester.

Generalization

It may be useful to generalize the meaning of events to include conditions that are not actually clock events. By setting one of the bits in the bit map that is not used as an actual clock event according to a condition detected by Data Access Table processing or by a local condition means it might be better if the server didn’t have to know about events itself.